XEROX Inter-Office Memorandum

To	Distribution	Date	July 6, 1972
From	E. R. Fiala	Location	Palo Alto
Subject	Revised MICRO document 9.2	Organization	PARC

The new version includes an appendix which outlines how to write microprograms using the MAXC microlanguage. This was not in 9.0 or 9.1.

ERF/jkf

Distribution:

C. Thacker
B. Lampson
P. Deutsch
H. Sturgis
C. Simonyi
L. Clark
E. McCreight
E. Fiala
R. Metcalfe
P. Heckel
Maxc Room

MICRO

MICROPROGRAM ASSEMBLER

MAXC 9.2

July 6, 1972

by

Edward Fiala
Peter Deutsch
Butler Lampson

Xerox Palo Alto Research Center
3180 Porter Drive
Palo Alto, CA 94304

MICRO

Microprogram Assembler

1. Introduction

This document describes the MICRO assembler for the MAXC microprocessor. It is designed to be used in conjunction with the MIDAS loader and debugger. Both of these programs are written in NOVA ALGOL, augmented by several machine language routines.

MICRO is a rather unspecialized one-pass assembler. It does not know anything specific about MAXC, but instead has a general facility for defining fields and memories [Although MICRO does not have any intrinsic memory definitions, MIDAS is assembled to expect certain memory names to be defined; consequently, MIDAS would have to be reassembled if memory names are used different from those it expects.], a standard string-oriented macro capability, and a rather unusual parsing algorithm which allows setting fields in memories in a natural way by defining suitable macros and neutrals with properly chosen names. MICRO also allows use of labels, literals, integer-valued variables (limited to 16 bits), and comments.

Microprogram preparation and debugging has the following procedures:

1) Prepare the program symbolics either using the NOVA text editor or in some other way, and enter them into the NOVA file system.

2) Assemble the symbolics using MICRO. Several different binary files may be created to reduce the amount of reassembly required during debugging. Each assembly requires the following dialog with the NOVA operating system:

```
MICRO/L/N/E  SYMTAB/R  SOURCE1 ... SOURCEN  BINARY/B  ERROR/E
LISTING/L
```

Global Flags	/L	produces an expanded listing of the input
	/N	suppresses binary and symbol table output
	/E	outputs errors to last named source file .ER
Local Flags	/R	recover from symbol table file.
	/L	puts expanded listing on named file
	/B	puts binary output and symbol table output on named file with extensions .MB and .ST, respectively.
	/E	put error listing on named file
	/S	puts symbol table on named file

At least one file name must appear with no flags: These are taken as source files and simply concatenated. If no binary file is specified, binary and symbol table output (unless suppressed

by /N) will go to the last named source file with extensions .MB and .ST, respectively; if no. listing file, last named file .LS; if no error file, the terminal (unless changed by global /E). Local flags override global ones. MICRO creates temporary files FMICRO.$$, IMICRO, and CMICRO and deletes them at the end of assembly. If you abort assembly with control-A, you will have to delete them yourself.

MICRO's binary output is generated in one pass and consists of a number of store directives to memories defined during the assembly followed by forward reference fixup directives, external address reference fixup directives, and new or changed address symbols for each memory.

3) Load the binary files created by MICRO with MIDAS. MIDAS provides facilities for loading the microprogram into the microprocessor, running it, inserting breakpoints, and examining and modifying the processor state. It also has facilities for calling MICRO to assemble individual statements, and for patching these over the previous contents of any memory location. Because expected speed of MICRO is about 25 lines/second when assembling declarations and about 2 lines/second when assembling microinstructions, assembly may be tedious, so considerable use of the patching facility is expected during debugging.

Note that the MICRO-MIDAS system has no provision for relocating a microprogram. However, the MICRO symbol table is dumped onto a file at the end of the assembly. Later, assembly can be continued at that point onto another binary output file. MIDAS can link up external address references after it loads the binary files. The MICRO symbol table from any assembly is available for use when patching. To avoid reassembling unchanged files (required when changes are not simply patches), one must partition his program into separate assemblies, each of which must use absolute location-counters for the various memories in the microprocessor. This may be difficult.

An expanded listing is produced only when either the global or local /L option is selected. Information put on the expanded listing is controlled independently for each memory by the LIST builtin which can enable/disable any of the following as described in section 3.9.2:

Numerically ordered list of address symbols;
Alphabetically ordered list of address symbols;
Memory stores in octal;
Memory stores by field.

Error messages produced by MICRO or by the ER builtin are printed on the listing file next to the memory store where they occur. Error messages are also output to the error file.

2. Program Structure

A source program consists of a string of characters delimited into comments and statements. All characters between two %'s are treated as a comment. All characters on a line following a * are a comment. It is intended that very long comments be enclosed in %'s while one-line comments be begun by *'s. The remaining text is delimited into statements by semicolons. There is no level of program structure superior to the statement (e.g., conditionals cannot span more than one statement).

All blanks, CR's, LF's, and non-printing characters are removed from source statements to avoid confusion between symbols like those below:

```
P+Q+1      P + Q + 1
RSH1       RSH 1
```

MICRO handles all code generation by table lookup and minimal use of conditionals. In particular, it does not evaluate P+Q+1 but rather looks it up in the symbol table. Since P + Q + 1 is the same for a human, we have chosen to suppress all blanks. Other non-printing characters are suppressed so that control characters don't appear invisibly in print names.

2.1 Statements

Statements are divided into clauses by commas and an indefinite number of clauses may appear in a statement. However, the size of MICRO's statement buffer limits statements to 300 characters at any one time. If this is exceeded at any time during the assembly of a statement, an error message is output. Examples of clauses are:

```
NAME,
NAME[ARG1, ARG2, ..., ARGN],
FOO¬FOO1¬FOO2¬P+Q+1
P ¬ STEMP
NAME[N1[N2[ARG]], ARG2]¬FOO[X].
```

P+Q+1 is a "source" while FOO, FOO1, and FOO2 are "destinations" or "sinks."

Note that names may be very long [a size limit is imposed by the 300-character length of the assembler's statement buffer], and that the only special characters are:

% and *	for comments as described above
[and]	for enclosing builtin, macrc, field, memory, and address argument lists
(and)	for causing nested evaluation

¬	as the final character of the token to its left
:	to put the address to its left into the symbol table with value equal to the current location and current memory
,	separates clauses or arguments
;	separates statements
#	#1, #2, etc., are the formal parameters inside macro definitions
Ø,1,2,3 4,5,6,7	are number components (all numbers in octal)

2.1.1 Tokens

The rules for delimiting clauses into tokens have been carefully chosen to permit the user of MICRO to write readable programs.

The parsing of statements is strictly right-to-left and the following definitions are required in explanation:

1) An L-token terminates the token to its left.
2) An R-token terminates the token to its right.

Then

(	R	group delimiter
)	L	group delimiter
[	L	builtin argument list delimiter
]		builtin argument list delimiter
,	LR	clause delimiter
:	LR	clause delimiter which takes the preceding token as an address in the current memory at the current address
¬	LR	separator which is part of the symbol to its left

Any text with an R-token to its left and an L-token to its right constitutes a token called a symbol whose meaning is determined by looking it up in the symbol table.

Text enclosed in parentheses is lexically independent of anything outside, and a parenthesized string of text is lexically equivalent to the "tail" which its evaluation produces. The following example clarifies this. In the expression:

```
FOO5(FOO1[FOO2]FOO3[FOO4])FOO6[FOO7]
```

the order in which expansions are recognized assuming that each FOO expansion leaves behind no text is:

```
FOO1[FOO2]
FOO3[FOO4]
```

```
F005F006[F007]
```

2.1.2 Tails

The handling of tails is a distinguishing peculiarity of MICRO. This works as follows. The tail is initialized to the nulstring at the start of processing a clause. When a neutral symbol is recognized using the rules for delimiting tokens (section 2.1.1), it is concatenated on the left of a string called the tail thusly:

```
temp ¬ concatenate (symbol, tail);
if tail = null do;
    tail ¬ temp;
else do;
    tail ¬ null;
    treat temp as a symbol;
end;
```

Parentheses push down the current tail and start a new null one. When the text inside is completely processed, its tail (null or neutral) is treated as though it were a string which had appeared without parentheses.

The use of neutral tails permits a very complicated machine like MAXC to be described by a relatively small number of macros and neutrals. The following example shows how this works.

MAXC has about 30 bus sources and 30 bus destinations, but not all combinations of source and destination are legal (a slow source may not feed a slow destination). An example using the bus is:

```
MDR¬X
```

X is a macro which expands to a store into the bus source field of the microinstruction and leaves behind the neutral symbol B. MDR¬ is the next token recognized. It is a macro which expands into a store into the bus destination field and leaves behind the neutral symbol B¬. B¬B is the next token recognized. Since the connection of a fast bus source to a fast bus destination is legal, B¬B has also been entered into the symbol table as a macro equivalent to the neutral symbol B. Thus the number of symbols which must be defined for describing bus sources and destinations is roughly 1/source plus 1/destination plus a small number of macros to describe legal connections of a class of sources to a class of destinations. Each class of objects is represented by a neutral symbol.

The innovative use of neutral symbols raises MICRO above a cloud of more complicated, inferior assemblers.

2.1.3 Clause Evaluation

When a clause is broken into top level tokens, the possible resulting symbol types and actions are given by the table below:

Symbol type	Action
undefined	See section 2.3
integer	Error message and abort clause expansion
address[clauselist]	Carry out a store of the word assembled by the clauselist at the location and memory of the address. and then increment the integer part of the address symbol.
address SYM	Replace by sourcemacro[SYM] (section 3.4)
address¬ SYM¬	Replace by sinkmacro[SYM] (section 3.4)
unbound address	Error message
memory[SYM,integer]	Create an address symbol "SYM" in the memory with value "integer"
field[address]	Perform field assignment (section 3.5)
field[integer]	Perform field assignment
field[undefined]	Generate forward reference for eventual field assignment at end of assembly or by MIDAS.
macro [args]	Expand it (section 3.1)
macro	Expand it
neutral	See section 2.1.2
neutral [args]	Error message
builtin [args]	Call the builtin function with arguments handled as discussed in Section 2.2

Ultimately, the original clause must reduce through macro and neutral expansions to a series of field assignments and builtin

calls with a neutral symbol in the "tail." The neutral symbol is then thrown away and the next clause is evaluated.

2.2 Treatment of Arguments

As briefly discussed in section 2.1.3, many symbol types may be followed by argument lists. The only difference among these is that fields, memories, addresses, and most builtins must be followed by an exact number of arguments. Macros, on the other hand, may have surplus arguments (ignored) or deficient arguments (nulstrings supplied). Conditionals may omit arguments (nulstrings supplied).

The nulstring argument is special in the following sense. If it appears where an integer result is wanted, it is equivalent to the value Ø; if it appears where a string is wanted, it is the nulstring; and, if it is looked up, it is undefined. Note that a MICRO program could define the nulstring as a symbol. It is not clear what would happen but MICRO does not check for this. ***Programmers beware!

Each builtin may choose one of three basic ways to receive its arguments: quoted, looked up in the symbol table, or evaluated. Some languages have a step short of evaluation which might be called "macro expansion", but MICRO does not make any distinction between macro expansion and complete evaluation of an argument. However, if a string of the form

```
NAME[arguments]:
```

occurs in a clause being evaluated, NAME[arguments] is expanded until a string is left without brackets or parentheses, and then this string is the one affected by the ":". However,

```
IFDEF[NAME[arguments], ...]
```

which looks up its first argument, will look up the entire string including the brackets. This is a limitation of MICRO which may someday be repaired. It prevents symbol names from being generated in some situations.

The exact meaning of "look up" and "evaluate" changes with the builtin. Those builtins which "lookup" an argument generally do so for a symbol type check or to decide what action to carry out based upon the symbol type. There is no way for macro definitions to get at symbol types. Only builtins can do this. This is an unfortunate limitation of MICRO.

Argument evaluation is slightly different from clause evaluation. For example, evaluating the argument for the field assignment

FNAME[VALUE] takes place as follows: evaluate the tokens in the argument right-to-left expanding all macros and neutrals, looking for one of the following:

1) An address. Use its integer part to complete the field assignments discussed in section 3.5.

2) An unbound address. Generate a forward reference.

3) An undefined symbol. Create an unbound address and generate a forward reference.

4) An integer. Complete the assignment as discussed in section 3.5.

If the argument is the nulstring, put the integer Ø into the field. If the argument is a neutral symbol, if any text is left when the address, integer, or undefined symbol is found, generate an error.

Note that a neutral symbol results in no error for clause evaluation, but an error for a field assignment while an integer results in an error in a clause but no error in an assignment.

Other builtins which evaluate their arguments may have different requirements. For example, the integer builtins SET and ADD (see section 3.8) accept only an integer result. Address[clauselist] evaluates the clauselist exactly as if it had occurred at the top level.

In all cases, if part of the argument being evaluated is in parentheses, that part is evaluated exactly as if it had occurred at the top level.

2.3 Undefined Symbols

The print-name of a symbol is a character string by which the symbol can be referred to in the source. When the lexical scan finds a string S of characters which is a symbol token (delimited by L or R-tokens), it looks for a symbol with print-name S. If no such symbol exists, an error is indicated except in the following cases:

2.3.1 Destination Addresses

S ends with ¬. In this case the ¬ is stripped off and the resulting string S' is looked up. If S' is found and has type address in memory MEM, S is replaced by MEMSINK[S'] as discussed in section 3.4.

2.3.2 Octal Numbers

S consists entirely of octal characters with an optional leading "-" sign. In this case it is treated like a symbol of type integer whose value is the octal number. Note that integers may not be larger than 15 bits in magnitude. (We realize this is an awful kludge.) Also note that it would be possible for an integer string to be entered into the symbol table, preempting the natural use of that integer. The assembler makes no check for this. ***Programmers beware!

2.3.3 Literals

S starts with an octal character. In this case it is split into a head OCT and a tail SYM such that OCT consists entirely of octal characters and SYM does not start with an octal character. The macro SYM is then called as described below:

The first argument of SYM is the four right-most octal characters. The second argument is the next four octal characters, and so on until the octal characters are used up. For example,

    37436521ØØØV

is replaced by

    V[1ØØØ, 3652, 374].

The awkwardness of the 16-bit limitation for integers is clearly pointed out by this kludge. Clearly V[37436521ØØØ] would have been much easier to work with and would have been possible if the integer size was greater than or equal to the memory size. Also, going from a three-integer 36-bit result back to a text string is made impractical by the integer size limit.

Note that literals cannot have a leading "-" sign.

3. Builtins

All of the predefined operations of MICRO are called builtins. With the exception of the BUILTIN builtin, none of them have *a priori* names but instead are assigned names by the programmer. Names are assigned to builtin operations by declaration statements of the form:

```
BUILTIN[BUILTIN,1];
```

where the second argument is the intrinsic operation number and the first argument is the name by which it is referred to. All builtins are called using this same syntax:

```
NAME[ARG1, ARG2, ..., ARG9];
```

The all-inclusive list of builtins is given in Table 1.

Note that the *only* print-name assembled into MICRO is BUILTIN (and error messages).

Builtin No.	Name	Discussion
1	BUILTIN	Section 3
2	M	Macros, section 3.1
3	N	Neutrals, section 3.2
4	MEMORY	Section 3.4
5	TARGET	Target memory, section 3.6
6	DEFAULT	Default value of memory bits, Section 3.7
7	F	Fields, section 3.5
10	PF	Field preassignments, section 3.5
11	SET	Integers, section 3.8
12	ADD	Section 3.8
13	IP	Integer part of an address, section 3.8
14	IFSE	Section 3.3
15	IFSET	Section 3.3
16	IFE	Section 3.3
17	IFG	Section 3.3
20	IFDEF	Section 3.3
21	IFME	Section 3.3
22	ER	Error messages, section 3.9.3
23	LIST	Section 3.9.2
24	INSERT	Section 3.9.1
25	NOT	Section 3.8
26	REPEAT	Section 3.10

Table 1: Builtins

Form	Condition
IFE[i1, i2, (true), (false)]	i1 = i2
IFG[i1, i2, (true), (false)]	i1 > i2
IFDEF[s1, (true), (false)]	s1 in symbol table and not unbound address
IFSE[s1, s2, (true), (false)]	s1 = s1
IFSET[field, (true), (false)]	any bit of field previously assigned
IFME[address, s1, (true), (false)]	memory name for address = string

Table 2: Conditionals

3.1 Macros

A symbol can be given a macro value by the clause

    M[NAME, body]

where the body is an arbitrary balanced string of characters (i.e., parentheses and brackets match up and are nested). Occurrences of the text

    #digit

in the body will be replaced by the corresponding actual parameters (counting left-to-right from 1) when the macro is called. Unsupplied arguments are nulstrings, surplus arguments are ignored, and #Ø will be replaced by the number of arguments supplied.

The lexical scan of a statement is done from right to left. Whenever a symbol S is detected, it is looked up. If S turns out to be a macro, then the macro body replaces both S and the bracketed argument list immediately to the right of S, if there is one. Thus after

    M[FOO, MUMBLE#1];

the text FOO[E]D; expands into MUMBLEED; note that D is not a symbol since] is not an R-token.

Note that the macro body is quoted and that MICRO has no provision for getting any part of it expanded at definition time.

3.2 Neutrals

A symbol which has been declared neutral by a clause of the form

    N[SYM]

is concatenated with the tail and handled as discussed in section 2.1.2.

3.3 Conditionals

There are a number of builtins which will substitute the text represented by one of their arguments if the other arguments meet some condition. These are called conditionals. The ones so far defined are given in Table 2, where (c1) is assembled only when the other arguments meet the indicated condition. A conditional and the argument list to its right are equivalent to the "true"

string, if the specified condition is met, or the "false" string, if it is not met. Note that any number of arguments may be omitted. The true and false strings may be any balanced strings of characters.

Although these conditionals can be used at the top level, they are intended for use inside macro definitions, and the string compare conditional could be used sensibly only inside macro definitions.

3.4 Memories, Addresses, and Stores

MEMORY[MEM, wordlength, length, sourcemacro, sinkmacro] causes creation of a memory. MICRO can manage a reasonable number (15) of these memories, subject to a 255-bit word-length limit and 32K-1 length limit. (Note that the display programs for the debugger currently require memories smaller than 36 bits to be defined as 36 bit memories in order that number columns get positioned right-justified.)

Once MEM has been defined, symbols can be defined as addresses in MEM and words of MEM can be initialized. An address ANAME in MEM is created by an expression of the form:

    MEM[ANAME, integer]

or by using

    ANAME:

in a clauselist which is stored in MEM. Stores into MEM are generated either by selecting an address in MEM as the target (see section 3.6) or by writing

    ANAME[(clauselist)]

which stores the word assembled by the clauselist into MEM at the location of the address ANAME and then increments ANAME. Note that the memory store and incrementing the address are done iff one or more field assignments result from the clauselist. Also note that the value of the word to be stored is initially determined by the DEFAULT statement for MEM (or Ø if there has been no DEFAULT statement). Finally, an error message will occur when one tries to store into an out-of-bounds address.

The use of sourcemacro MSRC and sinkmacro MSINK is discussed below.

The address ANAME has some peculiar properties. If it is evaluated as a token in a clauselist without a following argument list, it is replaced by the string

MSRC[ANAME].

If ANAME¬ appears and is undefined, it is replaced by

MSINK[ANAME].

Note, however, that forward and external references can be generated <u>only</u> in the context

FNAME[ANAME],

<u>not</u> when ANAME is used as a source or sink.

3.5 Fields, Assignments, and Preassignments

F[FNAME, leftbit, rightbit] causes a symbol of type field to be created. Leftbit and rightbit must evaluate to integers. Also, because of the NOVA's 16-bit integer size, the field should not be wider than 16 bits or else some bits of the field could never be set. Finally, leftbit must be in the range [Ø, 255] and rightbit in the range [leftbit, min (leftbit+15, 255)].

Clauses of the form

FNAME[integer];
FNAME[address]; or
FNAME[undefined];

where FNAME is a field, are used to construct memory words. A field assignment evaluates its argument in the manner discussed in section 2.2.

Field assignments also have the property that attempting more than one assignment to a field in a statement will cause an error unless the new value = old value. (When errors occur, the value ultimately left in a field is that of the final assignment to it.) Forward references do an FNAME[Ø] when they occur and fixup the true value later.

The preassignment PF[FNAME, integer] does nothing if any bits of field have previously been assigned. Otherwise, it is equivalent to FNAME[integer] except that a later assignment will overrule the preassignment and cause no error. Forward references are illegal in preassignments.

3.6 Target Memory

At any time TARGET[ANAME] will set the target address to ANAME which means that a statement of the form

    X: mumble;

where mumble must do at least one field assignment, is equivalent to:

    ANAME[(X: mumble)];

if at least one top level field assignment is done by mumble. Otherwise, the target has no effect. Note that the target memory is not preserved in the /R file and must be given again for each assembly.

3.7 Default Statement

Before assembly of a clauselist for storage into a memory MEM, the word is initialized to a value which may be overruled by the various assignments in the clauselist. Normally, the initial value is Ø, but this may be changed by the statement

    DEFAULT[MEM, (clauselist)].

Note that forward references are not permitted in the clauselist and that any of the default settings may be overruled by explicit assignments in the statement being assembled.

3.8 Integers: SET[INAME,I1], ADD[I1,I2,...,I8], NOT[I1], and IP[ANAME]

MICRO permits use of integer variables which are constrained (by the Nova's wordlength) to 15 bits in magnitude. SET[NAME,VALUE] looks up its first argument and evaluates its second with the following results:

Type of Name	Type of Value	Action
Undefined	Integer	Enter NAME in the symbol table with type integer and value VALUE.
Integer	Integer	Change the value of NAME to VALUE.

All other combinations are errors.

ADD[I1, I2, ..., I8], where all arguments are optional, produces an integer string result which is the sum of its arguments. Each of the arguments must evaluate to an integer. (Omitted arguments are Ø's).

Note that integer strings may begin with an optional "-" and must be in octal. However, the negative of an integer valued symbol cannot be obtained by inserting a leading "-"; -(ISYM) will not work, either. NOT[ISYM], providing the 1's complement of ISYM, can be used to get the negative by ADD[1, NOT[ISYM]].

IP[ANAME], where ANAME must be an address, is the integer part of the address. This must be done when an address is used in an arithmetic or set expression. (It is not reasonable to automatically take the integer part of an address because of confusion between its use as a source and its use as an integer).

Example: ADD[3, 4, 15]S is equivalent to 24S.

Other operations on integers can be added when there is a need for them. Boolean AND, OR, XOR, are possibilities.

3.9 Assembly Control

3.9.1 INSERT[filename]

INSERT causes the contents of the specified file to be inserted in the source text in place of the INSERT statement. This may be nested to a reasonable depth.

3.9.2 LIST[memory,mode]

LIST causes listing to be resumed or stopped to the listing file for all stores to the selected memory. At the end of assembly, LIST determines whether alphabetically and numerically ordered address symbols are output to each memory. The listing of each memory is controlled independently. Mode=Ø causes no listing of stores to the specified memory and no output of address symbols at the end of assembly. ANAME[(TAG:mumble)], where ANAME = 3Ø2 is listed in octal as:

```
302 (TAG) NNNN NNNN ... NNNN
```

if listing mode is 1 or as

```
302  (TAG) F1¬3, F2¬34, F3¬20;
```

if listing mode is 2, where F1, F2, and F3 represent <u>all</u> the <u>fields</u> to which <u>explicit</u> assignments were made during the assembly of (mumble). Fields which have non-zero values due to the action of a DEFAULT statement for the memory are not listed, nor are preassignments listed. Mode = 3 causes the store to be listed by both modes:

```
302 (TAG) NNNN NNNN ... NNNN
          F1¬3, F2¬34, F3¬20;
```

If the memory name is null, then the mode will be AND'ed with the memory mode for all memories other than the target, e.g. LIST[,Ø] suppresses listing of all non-target memories and LIST[,3] restores.

Adding 4 to the argument of LIST enables a numerically ordered list of address symbols at the end of assembly. Adding 1Ø enables an alphabetically ordered list of symbols.

Error messages are printed on the line after the listing of the memory word or between memory words if no field assignments were completed in the statement.

3.9.3 ER [message, stopcode]

ER is the builtin by which a MICRO program outputs messages to the error file (and to the listing file). Blanks are squeezed out of the message by the prescan so "-" signs or other printing characters should be used instead. The stopcode is Ø (if the second argument is the nulstring, then Ø is assumed) if the assembly should not be aborted or 1 if it should be aborted at this point. When the assembly is not aborted, the observations below are important.

Assembly of the statement in which the error occurred will continue from the point of the error. This may result in more error messages if the assembler gets confused.

The location counter gets incremented iff at least one store is done by the statement, so a statement with an error may still generate an output word, or it may not. If not, then all addresses will be off by 1 after the error.

3.10 Repeat Statement

REPEAT[il,TEXT] assembles TEXT il times. This is used primarily for initializing blocks of memory and duplicating instructions for processor diagnostics (e.g., testing each of the 256 values of SA). Since TEXT cannot include ";" stores to the target memory must be put in explicitly. In other words, the program cannot rely on the TARGET directive to insert "ILC[TEXT]" or whatever each time TEXT is repeated. Note that the statement buffer is cleared after each assembly of TEXT.

4. Error Messages Generated by Micro

Whenever an error condition is encountered by MICRO it outputs a meaningful message to both the error file (or terminal if there is no error file) and the listing file. In some cases MICRO plunges ahead, assembling the rest of the statement in error and then the rest of the source files. This may result in a number of strange error messages for a statement if the parser gets confused by an unmatched '(' or '[', an undefined symbol, or another condition. In other cases, when the assembly appears to be hopelessly confused, it is aborted. Also when the number of errors exceeds 2Ø the assembly is aborted.

All error messages, including those generated by the ER builtin, are of the form:

LASTTAG + NO.LINES [error message]

LASTTAG is the most recently defined address in the target memory and NO.LINES is the number of text lines in decimal (including comments) in the source program since LASTTAG. (LASTTAG is nul if no target has been defined or if no tags have occurred). This is done to facilitate finding the source line during editing later.

The error messages currently defined are listed below, in which the character @ should be replaced by the printname of the token related to the error. Unless marked otherwise, assembly continues from the error with no special action.

Note that the messages generated by MICRO can have blanks.

4.1 Program Organization Errors

SOURCE FILE @ DOES NOT EXIST[1]

COULD NOT OPEN FILE @ FOR 'INSERT'[1]

STORAGE FULL[1]

Storage required during the assembly is roughly proportional to the following computation:
1/2*Sum [namelength +1] for all symbols
+ 6* no. symbols
+ 1/2*Sum [length +1] of all macro definitions.
When this number is greater than the size of the buffer (approx. 13,500 NOVA words), the STORAGE FULL message results.

TOO MANY MEMORIES[1]

Limit is currently 15 memories

4.2 Declaration Errors

@ ALREADY DEFINED
The new definition will replace the old and this warning message will be printed.

MACRO @ REDEFINED
Just a warning (doesn't increment error count)

ARG NOT A MEMORY NAME[1]
For DEFAULT, which requires an argument to be of type memory

UNDEFINED SYMBOL @ IN 'DEFAULT'

BAD PARAMETERS FOR 'F'
A field may not be larger than 16 bits nor a memory wider than 256 bits, so rightbit >255 or rightbit-leftbit >16 are field definition errors

MEMORY @ ALREADY USED[1]

ILLEGAL WIDTH OR SIZE FOR 'MEMORY'[1]
Limits are 256 bits wide and 32K-1 in size

WRONG NO. ARGS FOR '@'
Only for those builtins which must have correct number of arguments. Macros may have too many or too few.

ILLEGAL BUILTIN NUMBER FOR '@'[1]

4.3 Statement Assembly Errors

END OF FILE INSIDE COMMENT
Terminates comment and forges ahead

INPUT STATEMENT TOO LONG
Maximum length is 300 characters. Text to the right of the 300th character is truncated.

STATEMENT TOO LONG
During macro expansion of the input statement, the unprocessed text is never permitted to exceed 300 characters. Text to the right is truncated.

MACRO ARGUMENT STORAGE FULL
Truncates characters right-to-left up to matching '[' and proceeds.

SYMBOL @ NOT LEGAL AS TOKEN
Symbol appears without its required argument list.

@ MAY NOT BE FOLLOWED BY [ ]
Only macros, builtins, fields, addresses, and memories may have '[' to their right

UNPAIRED) OR] IN ARGUMENT LIST

UNPAIRED)

UNPAIRED (

TOO MUCH NESTING OF ( ) AND [ ] IN CLAUSE
Limit is 8 levels

MISSING MACRO NAME OR TAG SYMBOL
No symbol to the left of a : or [.

MACRO '@' NOT DEFINED
Symbol to the left of a "[" wasn't defined

TAG @ ALREADY DEFINED

'TARGET' GIVEN AFTER FIELD SET[1]

NO TARGET FOR FIELD SET[1]

'TARGET' NOT LEGAL INSIDE A STORE[1]

@ UNDEFINED
Not including forward references. Plunges ahead with value Ø and type integer

FIELD @ DOES NOT FIT IN MEMORY @
Right bit of field > right bit of memory

VALUE @ DOES NOT FIT IN FIELD @
Left bits of value truncated before store

ARG IN FIELD STORE NOT INTEGER OR ADDRESS
Doesn't do field assignment and plunges ahead

FIELD @ ALREADY SET
The new value is stored into the field. This message will occur iff new value # old value.

ARG DOES NOT YIELD INTEGER VALUE
Assumes Ø and proceeds. Syntax OK but undefined symbol or address instead of integer.

BAD SYNTAX WHERE VALUE REQUIRED
Something complicated where a simple value expected

FIRST ARG OF 'PF' NOT FIELD
No action

FORWARD REFERENCE NOT LEGAL IN 'PF'
No action

STORE TO @ OUT OF RANGE FOR @

@ BAD FIRST ARG FOR 'SET'
Must be integer or undefined symbol. However, redefinition will take place.

INTEGER '@' TOO LARGE
Integer MOD 2**16 is used.

ARG NOT A FIELD NAME IN 'IFSET'

1. Aborts assembly.

5. Limitations of the Language

MICRO lacks some features and possesses certain limitations which make it less than the best language we might have implemented. These are discussed below:

1. It is not possible to relocate a microprogram at load time. The cost of non-relocatability is slowness rather than non-generality. (Since one can begin each successive assembly using the final symbol table from the preceding assembly, arbitrary relocation can be achieved by reassembling and reloading all the files. Even name conflicts are handled as generally as by a relocating assembler. Furthermore, a relocating assembler would have trouble merging constants from different assemblies at load time.)

2. Forward and external references are permitted only on field assignments which means that the occurence of

MDR¬STEMP, or STEMP¬MDR

where STEMP is an address in S, cannot be assembled if STEMP is a forward or external reference. Forward references to symbols which are not addresses are also impossible, but this restriction is not important to microprograms (?).

3. The storage available to the symbol table is limited. This has caused a microlanguage to be created which is less glorious than would otherwise have been possible. For example, instead of PQ RCY 1, one must write PQ RCY [1], because storage used by the 152 macros of the type PQ RCY 1 may result in crowding out other essential symbols. Furthermore, it is not possible to assemble the microcode for MAXC all at once. It has been necessary to partition the microcode into several segments, each of which begins with absolute location counters for the memories.

4. Integers larger than 16 bits are not handled. Fortunately there are no microinstruction fields larger than 11 bits in MAXC so that the integer size limit is an annoyance only in initializing other memories. A size limit of 36 bits would dodge these problems for MAXC, and a very large (approx. 256-bit) size limit would avoid the problem for any machine we are likely to build.

5. Assembly is extremely slow.

6. It is not possible to check the memory part of an address on forward or external references. Nor is it possible for programs to get at the type of a symbol, at the parameters of a field or memory, or at the name of the target memory. The 'lookup'

capability of builtins is not available through any language constructs.

7. It is not possible to have conditionals or macros which expand to more than one statement. Multi-statement conditionals would be useful for controlling assembly of blocks of code by parameters. Currently this must be done by manually inserting or deleting "%" - signs around such blocks.

8. It is not possible to pull print names apart or to construct print names except by using neutral subsymbols. In particular, it is not possible to construct constants parametrically such that, if a number of constants get the same value they will be the same location. This is true because one cannot generate the print name "142ØØØØS" (a literal) either directly from an integer or indirectly from the value assembled by assignments. (Note that if integers were large enough ADD[P1, P2, ... , P7]S <u>would</u> generate the literal in S.)

9. There is no way to expunge symbols. However, this is not an intrinsic problem since symbols may be redefined. The main use of expunging is to free up symbol storage, and this is valuable only because of the storage limitation already mentioned.

1Ø. There is no way to get macros expanded except by doing a complete evaluation nor is there a way to evaluate or expand part of a macro definition at definition time. Expanding part of a macro definition is only useful when expanding a macro that defines other macros, (and the macros defined in this way must not have arguments), so this capability is a convenience but not otherwise very important. However, there are a number of situations when part of an otherwise quoted argument wants to be expanded and there is no way to do this. For example,

```
IFDEF[FOO[E], (true clause), (false clause) ]
```

should lead to expansion of the macro FOO[E] before checking for a defined symbol.

11. It is not possible to make a print name such as "Ø" stand for an integer in one case and a print name in ancther. This means that expressicns such as

```
P¬Q¬Ø
```

and

```
FIELD1[Ø]
```

are not both possible. Effectively, symbols which are integers are unreasonable.

12. There is no way to get unpaired '(', '[', etc., into macro strings. (This has caused no problems so far. Maybe it is an advantage).

13. It is not possible to get blanks into user-defined error messages.

14. The basic mechanism of assembling statements right-to-left does not permit trying again when a particular choice of alternatives fails. For example, on MAXC some functions can be initiated by either F1 or F2 and the choice could depend on the assembly of the rest of the statement.

15. The REPEAT builtin should supply a ";" after each repetition of the text.

16. PF [field, value] was a bad choice because it makes parameterizing the values of a field impractical. For example, suppose that the function P¬P1 is accomplished by setting the PS field to 5Ø. What we would like to do is to define neutrals P¬ and P1 and then define the macro P¬P1 as PS[5Ø]. If the hardware is changed so that P¬P1 is accomplished by PS[2Ø] instead of PS[5Ø], we would prefer to change only the one macro P¬P1. However, there are also several instances of PF[PS,5Ø] which have to be found and changed and this is the reason why PF[field, value] was a bad choice. Instead, a preset-clauselist operation would have been better because then no other usage than P¬P1 would be needed.

To prevent some of the above limitations or to otherwise streamline or augment the language, the following changes should be considered (the ones followed by ? or ?? or ??? are not serious proposals).

1. Make integers at least 36 bits long for MAXC, and consider variable length integers. Currently, considerable inconvenience results from "making do" with 16-bit integers. Also this would make it possible to get the literal equivalent of a constant constructed from parameters, which would allow merging identically-valued constants.

2. Provide a builtin like the one for defining fields except that it takes an additional argument which is a memory name: AFIELD[AFNAME, leftbit, rightbit, memory]. AFNAME[address] works like FNAME[integer] except that its argument must expand to an address in "memory" rather than an integer, or if its argument is undefined, a forward address reference is assumed. Forward references to FNAME[undefined] would be illegal and

FNAME[address] would be illegal. Unbound addresses would contain the memory type. This would permit memory checking of addresses very conveniently (currently it is cumbersome) and would permit forward references to be checked also (??).

3. Multi-statement conditionals (and macro definitions ??) should be added.

4. It should be permissible for an argument list to appear to the right of a neutral symbol because of the following usage:

    P¬LB RSH [1]

where LBRSH is a neutral symbol, P¬ is a neutral symbol, and P¬LBRSH is a macro. The argument list [1] should be preserved until P¬LB RSH [1] is expanded.

5. In every place where an argument string is "looked up" for a builtin, all macros and neutrals should be expanded. In other words, "looking up" an argument should be identical to evaluating an argument, except that occurrence of any builtin causes an error. Expansion stops when a non-neutral non-macro symbol without brackets, parentheses, ¬, or : is left.

6. Currently <u>address¬</u> is handled by the assembler, but <u>undefined¬</u> and <u>macro¬</u> are not handled in any special way. Similarly, an undefined source is not handled. It might be useful to have these cases result in the substitutuicns UDEST[undefined], MDEST[macro] and USRC[undefined]. This would permit forward or external references to succeed where they don't currently and would permit macros which expand to addresses to be used. MDEST, UDEST, and USRC should be macrc names selectable by the programmer.

7. Currently the TARGET directive causes a top level statement to be equivalent to

    TARGLC[(#1)];

where #1 stands for the top level statement. This could be changed to a general macro whose first argument is the clauselist of the statement. However, this could result in slower assembly.

8. Instead of causing an error, integer results should be treated at the top level as neutral symbols equal to the octal text string for the integer. This would permit arithmetic to be performed and the result concatenated with text to select one of many macros or address symbols.

9. There are a number of places where a builtin could be used for its value as well as its function. For example, ADD[set[I1,2],1] could be used instead of (ADD[I1,2], SET[I1,1]). (???)

Appendix A. Binary Output Format

MICRO outputs binary memory images as a series of short blocks of 16-bit words. Each block begins with a word that specifies the type of the block; the number and format of following words depend on the block type.

During its pass through the source files, MICRO outputs a message to the file FMICRO.$$ whenever it encounters an assignment

    FNAME [NAME]

and NAME is undefined. At the end of processing the source files, MICRO reads back FMICRO.$$ and outputs either a type 3 or type 6 message (see below) to the binary file depending upon whether the symbol was a forward reference or undefined. Finally, it orders new or changed address symbols by memory and outputs them to the binary file. MIDAS can link up external address references at load time.

Address symbols for MIDAS to use in linking up external references are output as described below.

Type	Followed by	Use
Ø	nothing	Indicates the end of the binary file.
1	source line # (1 word); data (N words)	Specifies a data word to go in the current memory at the current location. The current location is to be incremented. N is just large enough to cover the width of the memory, and the value is left-justified, e.g., for a 36-bit memory N=3 and the first word goes in bits Ø-15, the second in 16-31, and bits Ø-3 of the third in 32-35. The source line # is zero if the word was generated by an INSERT file, and has bit Ø set if the word was generated in the main file by a STORE.

2	memory # (1 word); location (1 word)	Sets the current memory and the current location. Memory numbers are related to memory names by type 4 blocks (see below).
3	memory # (1 word); location (1 word); first bit * 256 + last bit (1 word); value (1 word)	Specifies a fixup. The value is to be stored into the given bits at the given location in the given memory. (Current memory and location settings are not affected.)
4	memory # (1 word); width of memory in bits (1 word); symbolic name of memory (L words)	Correlates a memory number with a user-supplied name. The name is packed 2 8-bit characters per word terminated by a null (all Ø) character; L=(C+2)/2 where C is the number of characters in the name. The type 4 block defining a memory will appear before any type 2 or 3 blocks storing into that memory.
5	memory # (1 word); value (1 word); first bit * 256 + last bit (1 word); address symbol name (L words)	Gives the definition of an address symbol. There is a type 5 block for every new or changed address symbol. All type 5 blocks appear together at the end of the binary file.

6	memory # (1 word); location (1 word); first bit * 256 + last bit (1 word); undefined symbol name (L words)	Specifies a reference to an undefined (external) symbol. The first three words have the same interpretation as for block type 3.

Appendix B. Basic Instructions for Preparing Microprograms for MAXC

We have made an effort to use the same names in the MICROPROCESSOR document (MAXC 8.2) as are used in the LANG file (Appendix C of this document). Consequently, we hope that once you are familiar with the terminology in the MICROPROCESSOR document, you will be able to write "natural" microprograms with the help of the general comments here and an occasional reference to Appendix C.

Generally speaking, a microinstruction for MAXC is prepared by stringing together a number of clauses in an arbitrary order. If it is possible to assemble a microinstruction to do those operations, the assembler will normally succeed in assembling it. Several rare exceptions are discussed later.

The clauses are of three types:

(1) A control clause:

```
GOTO[address, branch condition];
CALL[address, branch condition];
DGOTO[address, branch condition];
RETURN[branch condition];
```

If the branch condition is omitted always is assumed and if no control clause is provided, never is assumed. Note that simply saying:

```
RETURN
```

without any brackets is equivalent to

```
RETURN[ALWAYS].
```

(2) Routing data from a source or sources to a destination or destinations:

```
DEST2¬DEST1¬(SOURCE1)MRG(SOURCE2);
DEST¬SOURCE;
```

Where there are multiple sources or multiple destinations, it is possible to write several clauses instead of one compound clause. However, this is usually a mistake since it prevents the assembler from detecting some common errors and since the macros in LANG are capable of figuring out which microinstruction field should be used when several are possible.

(3) Performing a special function:

```
SIGNOVA
INCY
etc.
```

Note that the cycler functions for P and Q can be written either as:

```
PQ RCY [4]
 Q RSH 1
```

or as

```
P¬PQ RCY [4]
Q¬Q  RSH 1
```

In reading the MAXC microprocessor manual you will note that there are four fairly general function fields. Two of these are suggestively called bus source (BS) and bus destination (BD) while the other two are unsuggestively called function 1 (F1) and function 2 (F2). It is important to note that it is the <u>use</u> or operation initiated by a value in one of these fields that determines its syntax in the microlanguage, not the microinstruction field in which it appears. However, it is true that all values of BD use the destination syntax (NAME¬); all values of BS use the source syntax, while the values in F1 and F2 are split among source, destination, and special function syntaxes.

Some exceptions to arbitrary ordering of clauses which are quite infrequent are when one operation can be initiated by several methods but another operation cannot. In virtually all cases for MAXC, the LANG file will select the first unused field from the following sequence to initiate that operation:

BS bus source or BD bus destination;
F2 secondary function field;
F1 primary function field.

In practice this has only failed in three situations:

1. If you use INHINT (only a F2), make it the right-most clause of the statement.

2. If you use NPC¬ (only a F2), make it the right-most clause of the statement and if the "source" for NPC¬ is complex, split the clause, i.e.,

<u>Do not say</u>: NPC¬(MDR)MRG(SCRMEM)

because the assembler will use F2 to put SCRMEM on the bus.

<u>Instead say</u>: B¬(MDR)MRG(SCRMEM), NPC¬B;

However, this is a rare situation.

3. If you use multiple bus destinations put the scratch memory address left-most in the ¬ clause, e.g.,

```
SCRMEM¬MDR¬X   works but
MDR¬SCRMEM¬X   doesn't work.
```

LANG also contains macros to create address symbols in the various memories and to initialize data at those addresses. For example

```
RV[RFOO,3];
LV[LFOO,3];
SV[SFOO,3];
```

create variables in RM, LM, and SM respectively, with names RFOO, LFOO, and SFOO and with initial values of 3. Note that the initial value is Ø if the second argument is omitted.

The use of literals

```
3S
777R
21L
```

will create literals in SM, RM, and LM without prior declaration of these symbols as addresses.

To create a parameter FOO=2↑N, say

```
MP[FOO,N].
```

To create a parameter FOOBAR=sum of parameters FOO and BAR, say

```
SP[FOOBAR,FOO,BAR].
```

This works for six or fewer arguments to be summed.

To create a scratch memory constant whose value is the sum of six or fewer parameters, say

```
MC[CONST, P1,P2,P3,P4,P5,P6];
```

To create an address equal to a literal, say

```
SM[SFOO,IP[377S]];
LM[LFOO,IP[5L]];
RM[RFOO,IP[677R]];
```

This can be used to make one address syncnymous with another also.

Normally each assembly file begins with

TARGET[ILC]

where ILC is conventionally used as the locaticn counter for the microinstruction memory. Normally the macros for creating storage which we discussed above are used and the target memory is not changed.

```
BUILTIN[M,2];                  ◆DECLARE MACRO
BUILTIN[N,3];                  ◆DECLARE NEUTRAL
BUILTIN[MEMORY,4];             ◆DECLARE MEMORY
BUILTIN[TARGET,5];
BUILTIN[DEFAULT,6];
BUILTIN[F,7];                  ◆DECLARE FIELD
BUILTIN[FF,10];                ◆PREASSIGN VALUE TO FIELD
BUILTIN[SET,11];               ◆DECLARE INTEGER AND SET VALUE
BUILTIN[ADD,12];               ◆UP TO 8 INTEGER ARGUMENTS
BUILTIN[IP,13];                ◆INTEGER PART OF ADDRESS
BUILTIN[IFSE,14];              ◆IF STRING EQUAL
BUILTIN[IFSET,15];             ◆IF ANY BITS OF FIELD ASSIGNED
BUILTIN[IFE,16];               ◆IF INTEGERS EQUAL
BUILTIN[IFG,17];               ◆IF INTEGER 1 > INTEGER 2
BUILTIN[IFDEF,20];             ◆IF SYMBOL IN SYMBOL TABLE AND NOT
                               ◆UNBOUND ADDRESS
BUILTIN[IFME,21];              ◆IF MEMORY PART OF ADDRESS = STRING
BUILTIN[ER,22];                ◆TYPE ERROR MESSAGE
BUILTIN[LIST,23];              ◆SET LISTING MODE FOR MEMORY
BUILTIN[INSERT,24];            ◆INSERT FILE
BUILTIN[NOT,25];               ◆1'S COMPLEMENT
BUILTIN[REPEAT,26];            ◆REPEAT THE TEXT #2 #1 TIMES
```

```
%MEMORY DECLARATIONS MUST HAVE NAMES AND SIZES AGREEING WITH
THOSE IN MIDAS.

***CURRENTLY THE DEBUGGER REQUIRES MEMORIES < 36 BITS WIDE TO BE CREATED AS

***36-BIT MEMORIES.
***MEMORY NAMES MAY NOT BE BS AND BD NAMES.
%

*MEMORY[STK,13,14];                *STACK MEMORY NOT INITIALIZED BY ASSEMBLY
*MEMORY[MP,22,2000];               *MAP MEMORY NOT INITIALIZED BY ASSEMBLY
MEMORY[SM,44,1000,SSRC,SSINK];
MEMORY[DM,44,1000,DSRC,DSINK];
MEMORY[RM,44,40,RSRC,RSINK];
MEMORY[LM,44,40,LSRC,LSINK];
MEMORY[IM,110,4000,ISRC,ISINK];    *ISRC AND ISINK AREN'T DEFINED CURRENTLY

*PRINCIPAL LOCATION COUNTERS FOR MEMORIES

SM[SLC,0]; RM[RLC,10]; LM[LLC,20]; IM[ILC,30];


*INSTRUCTION MEMORY FIELD DEFINITIONS

F[BA,0,12];        *BRANCH ADDRESS
F[BT,13,14];       *BRANCH TYPE
F[BC,15,21];       *BRANCH CONDITION
F[LA,22,26];       *LEFT BANK ADDRESS
F[RA,27,33];       *RIGHT BANK ADDRESS
F[PS,34,41];       *P INPUT SELECT
F[QS,42,44];       *Q INPUT SELECT
F[AF,45,51];       *ALU FUNCTION SELECT
F[BS,52,56];       *BUS SOURCE
F[BD,57,63];       *BUS DESTINATION
F[F1,64,71];       *FUNCTION 1
F[F2,72,75];       *FUNCTION 2
F[SA,76,105];      *SCRATCH MEMORY ADDRESS
F[BRKP,106,106]; *BREAK-POINT BIT
F[SCP,107,107];   *SCOPE TRIGGER BIT
```

```
%BRANCH MACROS

"@" IS FOR A TYPE CHECK.  MACROS ACCEPTING BC ARGUMENTS
INSERT THE "@" IN FRONT OF THE NAME SUPPLIED BY THE PROGRAM.
IF THE BC ARG IS OMITTED, SET BC=ALWAYS.%

MC[CALL,ICHK[#1] BT[0] IFSE[#2,,@ALWAYS,@#2]];
MC[GOTO,ICHK[#1] BT[1] IFSE[#2,,@ALWAYS,@#2]];
MC[RETURN,BT[2] IFSE[#1,,@ALWAYS,@#1]];
MC[DGOTO,ICHK[#1] BT[3] IFSE[#2,,@ALWAYS,@#2]];

MC[ICHK,IFDEF[ZY#1,BA[ZY#1],BA[#1]]];


MC[ZY.-3,ADD[IP[ILC],-3]];
MC[ZY.-2,ADD[IP[ILC],-2]];
MC[ZY.-1,ADD[IP[ILC],-1]];
MC[ZY.,ILC];
MC[ZY.+1,ADD[IP[ILC],1]];
MC[ZY.+2,ADD[IP[ILC],2]];
MC[ZY.+3,ADD[IP[ILC],3]];


%BRANCH CONDITIONS

<BC+20> IS THE COMPLEMENT OF BC.  20 (NEVER) AND 0 (ALWAYS) ARE GIVEN
IMPLICITLY, NEVER EXPLICITLY.  THE DEFAULT FOR IM IS BC = NEVER.
GOTO, DGOTO, CALL, OR RETURN SET BC=ALWAYS IF NO BC ARGUMENT IS
WITH THE CALL.  THE EXTRA CHARACTERS "@" ARE PROVIDED SO THAT AN
ARGUMENT TYPE CHECK CAN BE MADE.%

MC[@ALWAYS,BC[0]];          MC[@NEVER,BC[20]];
MC[@ODD,BC[1]];             MC[@EVEN,BC[21]];
MC[@ALU8=0,BC[2]];          MC[@ALU8=1,BC[22]];
MC[@K=0,BC[3]];             MC[@K=1,BC[23]];
MC[@ALU<0,BC[4]];           MC[@ALU>=0,BC[24]];
MC[@H=0,BC[5]];             MC[@H=1,BC[25]];
MC[@X<0,BC[6]];             MC[@X>=0,BC[26]];
*7 IS N.I.
MC[@ALU=0,BC[10]];          MC[@ALU#0,BC[30]];
MC[@G=0,BC[11]];            MC[@G=1,BC[31]];
MC[@B<0,BC[12]];            MC[@B>=0,BC[32]];
*13 IS N.I.
MC[@ALU<=0,BC[14]];         MC[@ALU>0,BC[34]];
MC[@J=0,BC[15]];            MC[@J=1,BC[35]];
MC[@Y<0,BC[16]];            MC[@Y>=0,BC[36]];
*17 IS N.I.
```

```
*ALU FUNCTION SELECT

M[ P-1,AF[ 0 ]LU ];
M[ P-NJ,(AF[ 0 ],CJ&SJC)LU ];
M[ 2P,AF[ 3 ]LU ];
M[ 2P+1,(AF[ 3 ],CARRY1)LU ];
M[ 2P+J,(AF[ 3 ],CJ&SJC) ];
M[ P AND NOT Q - 1,AF[ 4 ]LU ];
M[ P-Q-1,AF[ 6 ]LU ];
M[ P + NOT Q,AF[ 6 ]LU ];
M[ P-Q-NJ,(AF[ 6 ],CJ&SJC)LU ];
M[ P-Q,(AF[ 6 ],CARRY1)LU ];
M[ P AND NOT Q + P,AF[ 7 ]LU ];
M[ P AND Q - 1,AF[ 10 ]LU ];
M[ P + Q,AF[ 11 ]LU ];
M[ P+Q+1,(AF[ 11 ], CARRY1)LU ];
M[ P+Q+J,(AF[ 11 ], CJ&SJC)LU ];
M[ P AND Q + P,AF[ 13 ]LU ];
M[ P+1,(AF[ 17 ], CARRY1)LU ];
M[ P+J,(AF[ 17 ], CJ&SJC)LU ];

M[ NOT P,AF[ 20 ]LU ];
M[ NOT P AND Q,AF[ 21 ]LU ];
M[ NOT P AND NOT Q,AF[ 22 ]LU ];
M[ A0,AF[ 23 ]LU ];                  ***0 NOT POSSIBLE--CONFUSED WITH INTEGER
M[ NOT P OR Q,AF[ 24 ]LU ];
M[ AQ,AF[ 25 ]LU ];                  *** Q IS AMBIGUOUS AND IS DEFINED WITH BUS
                                     ***SOURCES
M[ P=Q,AF[ 26 ]LU ];
M[ P AND Q,AF[ 27 ]LU ];
M[ NOT P OR NOT Q,AF[ 30 ]LU ];
M[ P#Q,AF[ 31 ]LU ];
M[ NOT Q,AF[ 32 ]LU ];
M[ P AND NOT Q,AF[ 33 ]LU ];
M[ A1,AF[ 34 ]LU ];                  *** -1 NOT POSSIBLE--CONFUSED WITH INTEGER
M[ P OR Q,AF[ 35 ]LU ];
M[ P OR NOT Q,AF[ 36 ]LU ];
M[ P,AF[ 37 ]LU ];

N[ LU ]; N[ Q ];              *THE ALU AND THE Q-REGISTER ARE OBJECT CLASSES
```

```
%THE LEGAL CYCLER STATEMENTS ARE:
    PQ RCY [Y]              WHERE 0 <= [Y] <= 44 OR 6 GETS SET
    PQ RCY [44-Y]
    PQ RCY [INTEGER]        WHERE 0 <= INTEGER <= 46
    PQ LCY [INTEGER]        WHERE 0 <= INTEGER <= 3
    LB LSH [INTEGER]        WHERE 0 <= INTEGER <= 3
    LB RSH [INTEGER]
    RB LSH [INTEGER]
    RB RSH [INTEGER]
AND ALSO EACH OF THE ABOVE WITH "PQ" REPLACED BY "OQ", "QQ", OR "NOTLUQ".
%

M[PQ RCY, IFSE[#1,Y,PS[76],IFSE[#1,44-Y,PS[75],PS[#1]CHK[46,#1]]]CY?];
M[QQ RCY,(PQ RCY [#1], RCYQQ)];
M[OQ RCY,(PQ RCY [#1], RCYOQ)];
M[NOTLUQ RCY,F1[47] PQ RCY [#1]];

M[PQ LCY,PS[ADD[74,-#1]]CHK[3,#1]CY?];
M[QQ LCY,(PQ LCY [#1], RCYQQ)];
M[OQ LCY,(PQ LCY [#1], RCYOQ)];
M[NOTLUQ LCY,F1[47] PQ LCY [#1]];

*MACRO TO GIVE ERROR IF #2 > #1 OR #2 > 0.

M[CHK,IFG[#2,#1,ER[CY?],IFG[0,#2,ER[CY?]]]];

N[LB]; N[RB]; N[P←]; N[P1];
N[CY?]; N[P←CY?];

M[P←LB,PS[56]LB];
M[P←RB,PS[65]RB];

M[LB LSH,PS[ADD[56,-#1]]CHK[3,#1]CY?];
M[LB RSH,PS[ADD[56,#1]]CHK[3,#1]CY?];
M[RB LSH,PS[ADD[65,-#1]]CHK[3,#1]CY?];
M[RB RSH,PS[ADD[65,#1]]CHK[3,#1]CY?];

M[P←B,PS[47]B];
M[P←SS,PS[47]SS];
M[P←SMA,PS[47]SB];
M[P←I,I(PS[47]' BS[7],DGOTO[.+1])];   *P IS THE ONLY LEGAL SINK FOR I
M[P←P1,PS[50]?];
M[P←LU,PS[51]LU];
M[LU ARSHC 1,PS[52]CY?];

M[P←Q,PS[44]Q];                       *USES THE CYCLER PQ RCY [44] BUT
                                      ***BEWARE IF RCYQQ, RCYOQ, OR RCY NOTALUQ
```

```
*Q INPUT SELECTION

M[Q←LB,QS[0]LB];
M[Q←RB,QS[4]RB];
M[Q←LU,QS[2]LU];
M[Q←B,QS[7]B];
M[Q←SS,QS[7]SS];
M[Q←SMA,QS[7]SB];
M[Q←Q,QS[5]Q];
M[RB ARSHC 1,QF?(QS[6],LU ARSHC 1)];
M[RB RCY 1,QF(QS[6] RCYQQ)];   ***TOO CLEVER? (ALSO SEE BELOW)
M[RB RSH 1,QS[6]QF?];

M[Q RSH 1,QS[3]QF?];
M[Q RCY 1,QF(QS[3] RCYQQ)];
M[Q ARSHC 1,QF(QS[3] ASHOVF)];
M[Q LSH 1,QS[1]QF?];
M[Q LCY 1,QF(QS[1] RCYQQ)];

M[Q←]; M[QF?]; M[Q←QF?];
```

```
MC?];          ♦FOR ERROR CHECK WHEN F1 OR F2 USED AS A SOURCE

MCSMA-AS-SA??];                    ♦FOR ERROR CHECK WHEN NON-SM ADDRESS USED

♦FUNCTIONS REALIZED BY BOTH F1 AND F2

MCNOF2,F2[10]];                    ♦NO FUNCTION
MCSETSF,IFSET[F2,F1[41],F2[11]](#1)-AS-SA??];
♦♦♦NOTE THAT KNEWCOMM IS ALSO F1[60] AND F2[5]
♦READS=F1[60]=F2[5]=KNEWCOMM NEVER GIVEN EXPLICITLY, ONLY USED BY SB,
♦ASB, AND QLS MACROS.
MCKNEWCOMM,IFSET[F2,F1[60],F2[5]]];
♦LOADS=F1[61]=F2[6] NEVER GIVEN EXPLICITLY, ONLY USED BY SSINK

♦NOTE THAT RCY00 AND RCY00 PREVENT US FROM USING PO RCY 0 AS THE DEFAULT
♦LOADING RULE FOR THE P REGISTER.  HENCE CHANGE TO P♦F1.

MCRCY00,IFSET[F2,F1[46],F2[12]] PF[PS,50]?];
MCRCY00,IFSET[F2,F1[50],F2[15]] PF[PS,50]?];
MCCARRY1,IFSET[F2,F1[42],F2[13]]?];
MCPOP,IFSET[F2,F1[67],F2[16]]?];
MCWRESTART,IFSET[F2,F1[14],F2[7]]?];


♦F2 ONLY

MCASHOVF,F2[14]?];
MCACFS,F2[17]?];
MCINHINT,F2[0]?];
MCNPC♦,F2[1]B♦];                   ♦AUTOMATICALLY INHIBITS INTERRUPTS
```

```
♦FUNCTIONS REALIZED ONLY BY F1

M[NOF1,F1[0]];

M[IREF,F1[1]?];
M[BIREF,F1[2]?];
M[RREF,F1[3]?];
M[RREFDXK,F1[4]?];
M[RMWREF,F1[5]PF[F2,0]];    ♦INHINT NOT FORCED BECAUSE OF DGOTO, NPC♦
M[RMWREFDXK,F1[6]PF[F2,0]]; ♦INHINT NOT FORCED BECAUSE OF DGOTO, NPC♦
M[WREF,F1[7]?];
M[WREFDXK,F1[10]?];
M[XREF,F1[11]?];
♦M[MAR←,F1[12]B←];   ♦♦THIS FUNCTION DELETED
M[MDRL←,F1[13]B←];
♦WRESTART ALSO A F2

♦M[KMAR←,F1[15]B←];   ♦♦THIS FUNCTION DELETED
M[KMDRL←,F1[16]B←];
M[KWRESTART,F1[17]?];
M[KRDATA,F1[20]SS];
M[KWDATA,F1[21]B];

M[SIGNOVA,F1[22]?];

M[INCY,F1[23]?];
M[DECY,F1[24]?];
M[NEGY,F1[25]?];
M[YKPTR←,F1[26]B←];
♦INC4Y WAS INCORPORATED IN MAP4♦

M[INCX,F1[30]?];
M[DECX,F1[31]?];

M[INCAC,F1[32]?];
M[DECAC,F1[33]?];

M[SETF,F1[34](#1)-AS-SA??];
M[SETFC,(F1[35] PF[BA,ZY.+1] @#2)(#1)-AS-SA??];
M[CLEARF,F1[36](#1)-AS-SA??];
M[CLEARFC,(F1[37] PF[BA,ZY.+1] @#2)(#1)-AS-SA??];
M[SETFB,(F1[40] PF[BA,ZY.+1] @#2)(#1)-AS-SA??];
♦SETSF IS ALSO A F2

♦CARRY1 IS ALSO A F2
M[CJ&SJC,F1[43]?];
M[SETHOVF,F1[44]?];
M[SETOVPC01,F1[45]?];
♦RCY00 IS ALSO A F2
♦RCYNOTALU@ USED ONLY IMPLICITLY BY NOT(ALU FUNCTION)@ RCY [INTEGER]
♦AND NOT(ALU FUNCTION)@ LCY [INTEGER].
♦RCY00 IS ALSO A F2

♦DEFAULT P3 IS P0 RCY [0] WHICH  DOES NOT CHANGE P UNLESS RCY00 OR RCY00
♦OCCURS, IN WHICH CASE CHANGE TO P←P1.
♦♦♦INTERRUPT ROUTINES BEWARE OF CLOBBERING P WHEN USING RCY00 OR RCY00.
```

```
M[LDPALUH,F1[51]?];                 ←DON'T LOAD P IF ALU0=H
M[035ALUG,F1[52]?];                 ←035←(ALU0#G) IF Q LSH 1
←READALU=F1[53] ALWAYS USED IMPLICITLY

%THE TWO FUNCTIONS BELOW GENERATE NEGATIVE NUMBERS WHICH ARE TRUNCATED BY
ADDITION OVERFLOW SO THAT THEY WILL FIT IN THE MICROINSTRUCTION FIELD.
THE ADDITION DOES NOT CHANGE THE LOW SIX BITS OF VALUE WHICH IS ALL THAT
IS RELEVANT HERE.%
M[SAMASK,F1[55] SA[ADD[NOT[#1],100]] ?];
M[BAMASK,F1[56] BA[ADD[NOT[#1],100]] ?];
M[AMASK,F1[57] AF[#1] ?];
M[XMASK,F1[54]?];

←READS ALSO A F2 AND A BUS SOURCE
←KNEWCOMM=READS ALSO A F2 BUT NOT A BUS SOURCE
←LOADS ALSO A F2 AND A BUS DESTINATION

M[ARM←,F1[62]B←];                     ←ARM[20,35]←B[20,35]
M[ARM,F1[63]B];                       ←B[20,35]←ARM, B[14,17]←INTNO, B[0]←INT

M[PREIRET,F1[64]?];
M[IRET,F1[65]?];

M[FRZBALUBC,F1[66]?];
←POP ALSO A F2
```

%BUS CONNECTION MACROS

EACH CLASS OF OBJECTS FOR THE MICROPROCESSOR IS REPRESENTED BY A NEUTRAL SYMBOL, AND THE LEGAL CONNECTIONS OF THESE CLASSES (BY "←" OR BY "MRG") ARE REPRESENTED BY MACROS WHICH REDUCE TO A NEUTRAL SYMBOL. ILLEGAL CONNECTIONS OF THESE OBJECTS ARE NOT DEFINED AND CAUSE ERRORS.

CONNECTION TO THE BUS IS MOST COMPLICATED BY THE LARGE NUMBER OF COMBINATIONS OF OBJECT CLASSES WHICH CONNECT TO THE BUS. THESE ARE LISTED BELOW:

LU	THE ALU LEAVES THIS NEUTRAL SYMBOL BEHIND
SMA	SCRATCH MEMORY ADDRESSES LEAVE THIS NEUTRAL SYMBOL BEHIND
SS	SLOW SOURCES
B	FAST SOURCES
Q	Q REGISTER (SEPARATE CLASS BECAUSE OF AMBIGUOUS ROUTINGS)
I	THE INSTRUCTION MEMORY
P←	P REGISTER
Q←	Q REGISTER
B←	FAST DESTINATIONS
SD←	SLOW DESTINATIONS

THE CONNECTION MACROS HAVE TO BE FAIRLY COMPLICATED BECAUSE, WHILE SOME CONNECTIONS CAN ONLY BE ACCOMPLISHED IN ONE WAY, OTHERS CAN BE ACCOMPLISHED IN SEVERAL. FOR EXAMPLE, THE SCRATCH MEMORY CAN BE CONNECTED AS A BUS SOURCE USING EITHER THE BS, F1, OR F2 FIELDS. IT CAN BE CONNECTED AS A BUS DESTINATION USING EITHER THE BD, F1, OR F2 FIELDS. ALSO, THE ALU CAN BE CONNECTED TO THE BUS USING EITHER BS OR F1, AND THE Q-REGISTER CAN BE CONNECTED TO THE BUS EITHER BY ROUTING IT THROUGH THE ALU OR DIRECTLY. NEARLY ALL THE OTHER BUS-CONNECTED OBJECTS CAN BE CONNECTED TO THE BUS IN ONLY ONE WAY (EITHER BY BS, F1, OR BD).
%

```
M[B]; M[B←]; M[SD←]; M[SS]; M[SMA];   ♦OBJECT CLASSES

M[B←B,B];                    M[B←Q,BS[24]Q];
M[SD←B,B];                   M[SD←Q,BS[24]Q];
M[B←SS,SS];                  M[B←LU,BS[25]LU];
M[B←SMA,BS[6]SS];
♦SD←SS IS ILLEGAL


♦MULTIPLE SOURCE REDUCTION MACROS

M[B MRG SMA, SB];
M[SS MRG SMA, SB];
M[LU MRG SMA, ASB];
M[B MRG LU, AB];
M[SS MRG LU, AB];
M[SMA MRG LU, ASB];
M[SMA MRG B, SB];
M[SMA MRG SS,SB];
M[LU MRG B, AB];
M[LU MRG SS, AB];
M[Q MRG SMA,BS[24]SB];
M[SMA MRG Q,BS[24]SB];
M[Q MRG LU,BS[24]AB];
M[LU MRG Q,BS[24]AB];
```

```
M[Q MRG B,QB];
M[B MRG Q,QB];
M[Q MRG SS,SS(,QB)];
M[SS MRG Q,SS(,QB)];
M[Q MRG B MRG SMA,QBS];
M[Q MRG SMA MRG B,QBS];
M[B MRG Q MRG SMA,QBS];
M[B MRG SMA MRG Q,QBS];
M[SMA MRG B MRG Q,QBS];
M[SMA MRG Q MRG B,QBS];
M[Q MRG SS MRG SMA,QBS];
M[Q MRG SMA MRG SS,QBS];
M[SMA MRG Q MRG SS,QBS];
M[SMA MRG SS MRG Q,QBS];
M[SS MRG Q MRG SMA,QBS];
M[SS MRG SMA MRG Q,QBS];
M[Q MRG LU MRG SMA,QLS];
M[Q MRG SMA MRG LU,QLS];
M[LU MRG Q MRG SMA,QLS];
M[LU MRG SMA MRG Q,QLS];
M[SMA MRG Q MRG LU,QLS];
M[SMA MRG LU MRG Q,QLS];
M[B MRG SMA MRG LU,ASB];
M[SS MRG SMA MRG LU,ASB];
M[B MRG LU MRG SMA,ASB];
M[SS MRG LU MRG SMA,ASB];
M[SMA MRG B MRG LU,ASB];
M[SMA MRG LU MRG B,ASB];
M[LU MRG B MRG SMA, ASB];
M[LU MRG SMA MRG B,ASB];
M[SMA MRG SS MRG LU,ASB];
M[SMA MRG LU MRG SS,ASB];
M[LU MRG SS MRG SMA,ASB];
M[LU MRG SMA MRG SS,ASB];
M[B MRG B,B];
M[B MRG SS,SS];
M[SS MRG B,SS];
M[SS MRG SS,SS];

M[QBS,IFSET[BS,ASB(,AQ),BS[24]SB]];
M[QLS,BS[24]F1[53]F2[5]SS];
M[SB,IFSET[BS,IFSET[F2,F1[60],F2[5]],BS[6]]SS];
M[AB,IFSET[BS,F1[53],BS[25]]SS];
M[ASB,IFSET[BS,F1[53]F2[5]SS,BS[25]SB]];
M[QB,IFSET[BS,AB(,AQ),BS[24]B]];
```

```
*BUS SOURCES AND DESTINATIONS

M[NULL,BS[0]B];                  *DEFAULT BUS DESTINATION IS 0, NEVER GIVEN
                                 *EXPLICITLY
M[X,BS[1]B];                M[X←,BD[1]B←];
M[Y,BS[2]B];                M[Y←,BD[2]B←];
M[AC,BS[3]B];               M[AC←,BD[3]B←];
M[MAP,BS[4]SS];             M[MAP←,BD[4]SD←];
M[D,BS[5]SS];               M[D←,BD[5]SD←];

*** NO ERROR CHECK FOR BOTH DM AND SM USED ' SAME INSTRUC' ON
*** FORWARD REFERENCES IMPOSSIBLE FOR ADDRE.. SOURCES

M[SSRC,SA[*1]SMA];

*NOTE THAT AN SM ADDRESS SHOULD BE TO THE LEFT OF A BUS DESTINATION IN
*A MULTIPLE LEFT-ARROW EXPRESSION

M[SSINK, IFSET[BD, IFSET[F2, F1[61], F2[6]], BD[6]] SA[*1] SD←];

M[I];   %ONLY INTO P%       M[I←,SD←(BD[7],DGOTO[.+1])];

M[MDR,BS[10]B];             M[MDR←,BD[10]B←];
M[MDRL,BS[11]B];            M[READ←,BD[11]B←];
M[MAR,BS[12]B];             M[RMW←,B←(BD[12],PF[F2,0])];
                            M[WRITE←,BD[13]B←];

M[KMDR,BS[14]B];            M[KMDR←,BD[14]B←];
M[KMDRL,BS[15]B];           M[KREAD←,BD[15]B←];
M[KMAR,BS[16]B];            M[KWRITE←,BD[16]B←];
                            M[KRMW←,BD[17]B←];

M[KUNIT,BS[20]SS];          M[KUNIT←,BD[20]B←];
M[KSTAT,BS[21]SS];          M[KSET←,BD[21]B←];
                            M[KCSET←,BD[22]B←];

M[NOTF,BS[23]SS];           M[ISPLIT←,BD[23]B←];
M[BQ,BS[24]B];              M[FSPLIT←,BD[24]B←];
                            M[BSPLIT←,BD[25]B←];

M[STACK,BS[26]B];           M[STACK←,BD[26]B←];
M[NPC,BS[27]B];             M[MAPVA←,BD[27]B←];
                            M[MAP4←,BD[30]SD←];
M[XTOP,BS[31]B];            M[XSPLIT←,BD[31]B←];
                            M[YSHIFT←,BD[32]B←];
```

```
%INSTRUCTION MEMORY DEFAULT STATEMENT.  EVERYTHING REMAINS UNCHANGED
EXCEPT THE ALU AND BUS BRANCH CONDITIONS WHICH ARE =0 AND >=0.%

DEFAULT[IM,<GOTO[ILC,NEVER],Q←0,A0 RCY[0],LA[4],RA[4],A0,SA[0],
    B←NULL,NOF2,NOF1,BD[0]>];


%MACRO FOR INITIALIZING INSTRUCTION IN OTHER THAN THE TARGET LOCATION:
    AINT[INTEGER, <CLAUSELIST>];
STORES THE CLAUSELIST IN IM AT LOCATION INTEGER.%

M[AINT,<XSLC[#2],IM[XSLC,#1]>];
```

```
*FIELDS FOR INITIALIZING 36-BIT WIDE MEMORIES

F[E3,0,13]; F[E2,14,27]; F[E1,30,43];

*MACRO TO INITIALIZE (36-BIT) VARIABLES IN THE TARGET MEMORY.  THIS IS
*DONE BY WRITING 432156732100V (I.E., AS A LITERAL).

M[V,IFSE[#4,,E1[#1] E2[#2] E3[#3], ER[#4#3#2#1V???]]];


*SPECIAL SCRATCH MEMORY STUFF

SM[SY,0]; SM[SY1,1];

M[SFULL,IFG[IP[SLC],377,ER[S-FULL-AT-#1]]];

M[S,#3#2#1S(SLC[#3#2#1S:#4#3#2#1V],SFULL[#3#2#1S])]; *LITERALS IN SM

M[SV,SLC[#1:#2V] SFULL[#1]];   *VARIABLES IN SM

%MACROS TO CREATE CONSTANTS IN S.  SEVERAL PROPERTIES ARE DESIRABLE:

    1)  IF A LITERAL OF THE SAME VALUE HAS BEEN PUT IN S, POINT
        THE CONSTANT AT THE SAME LOCATION.
    2)  OTHERWISE, DEFINE THE CONSTANT AS AN ADDRESS IN S AND
        INITIALIZE THAT LOCATION TO THE CONSTANT VALUE.
    3)  ALL CONSTANTS OF THE SAME VALUE SHOULD WIND UP POINTING AT
        THE SAME LOCATION.  THIS MEANS THAT THE LITERAL EQUIVALENT OF
        THE SYMBOL SHOULD ALSO BE PUT INTO THE SYMBOL TABLE AND THAT
        THE PROCESS OF DEFINING A CONSTANT SHOULD INCLUDE CHECKING
        FOR PREVIOUS ENTRY OF THE LITERAL EQUIVALENT INTO THE SYMBOL TABLE.
    4)  IT WOULD ALSO BE DESIRABLE TO HAVE SOME CHECKS FOR THE
        MISTAKEN USE OF THE CONSTANT AS THE ARGUMENT TO LOADS OR
        AS A BUS DESTINATION SINCE CONSTANTS SHOULD NOT BE DIRTIED
        BY STORES.
    5)  ABILITY TO CONSTRUCT THE CONSTANT IN A NATURAL WAY FROM
        MORE BASIC PARAMETERS.  FOR EXAMPLE, IT IS IMPORTANT TO BE
        ABLE TO DEFINE THE BITS IN THE F REGISTER SYMBOLICALLY
        (BECAUSE THESE ASSIGNMENTS HAVE BEEN VOLATILE IN THE
        PAST AND ARE LIKELY TO BE VOLATILE IN THE FUTURE) AND
        TO DEFINE EVERY AGGREGATE OF THESE FLAGS FROM THE MORE FUNDA-
        MENTAL DEFINITIONS.

AT THE MOMENT (1), (3), AND (4) ABOVE ARE NOT IMPLEMENTED.  SOME
FACILITIES FOR MULTI-WORD INTEGER ARITHMETIC AND FOR CONVERTING
INTEGERS BACK INTO TEXT STRINGS IS REQUIRED.

MEANWHILE, FOUR MACROS ARE IMPLEMENTED:
   PM[NAME,OCTALSTRING] MAKES A PARAMETER OF NAME;
   MP[NAME,INTEGER] MAKES A ONE-BIT PARAMETER OF NAME WITH A ONE IN THE BIT

SELECTED BY INTEGER;
   SP[NAME,P1,P2,P3,P4,P5,P6] MAKES A PARAMETER NAME EQUAL TO THE SUM
OF PARAMETERS P1, P2, P3, P4, P5, AND P6;
   MC[NAME,P1,P2,P3,P4,P5,P6] MAKES AN SM CONSTANT OF NAME FROM
PARAMETERS P1, P2, P3, P4, P5, AND P6;
   SM[NAME,IP[LITERAL IN S]] MAKES AN SM CONSTANT OF NAME AND ALSO
```

```
DEFINES THE LITERAL.
THE PARAMETER "NAME" IS DEFINED BY THE INTEGERS NAME. (BITS[30-43]),
NAME! (BITS[14-27]), AND NAME@ (BITS[0-13]).
%


*DEFINE 36-BIT PARAMETER

M[PM,(IFE[TT.,,,SET[#1.,TT.]] IFE[TT!,,,SET[#1!,TT!]]
    IFE[TT@,,,SET[#1@,TT@]],#2PMM)];
M[PMM,(SET[TT.,#1], SET[TT!,#2], SET[TT@,#3])];

*DEFINE A ONE-BIT PARAMETER

M[BX,B#1];
M[MP,IFG[#2,27,SET[#1.,ADD[#2,-30]BX],IFG[#2,13,SET[#1!,ADD[#2,-14]BX],
    SET[#1@,ADD[#2]BX]]]];

SET[B0000,4000];
SET[B0001,2000];
SET[B0002,1000];
SET[B0003,400];
SET[B0004,200];
SET[B0005,100];
SET[B0006,40];
SET[B0007,20];
SET[B0010,10];
SET[B0011,4];
SET[B0012,2];
SET[B0013,1];

*0 IF UNDEFINED, #1 OTHERWISE

M[RZ,IFDEF[#1,#1]];
M[AZ,SET[TT#1,ADD[RZ[#2.#1],RZ[#3.#1],RZ[#4.#1],RZ[#5.#1],RZ[#6.#1],
    RZ[#7.#1]]]];     *ADD UP PART OF A PARAMETER
M[TZ,AZ[.,#1,#2,#3,#4,#5,#6]AZ[!,#1,#2,#3,#4,#5,#6]AZ[@,#1,#2,#3,#4,#5,#6]]


*MAKE A PARAMETER FROM PARAMETERS

M[SP,IFG[#0,7,ER[TOO.MANY.ARGS.FOR.#1]]
     TZ[#2,#3,#4,#5,#6,#7] IFE[TT.,,,SET[#1.,TT.]]
     IFE[TT!,,,SET[#1!,TT!]] IFE[TT@,,,SET[#1@,TT@]]];


*MAKE AN SM CONSTANT FROM PARAMETERS

M[HMC,IFDEF[#1,ER[#1-PREVIOUSLY-DEFINED],SFULL[#1]
    IFG[#3,7,ER[TOO-MANY-ARGS-FOR-#1],SLC[#1:E1[#2[TT.]] E2[#2[TT!]]
    E3[#2[TT@]]]]]]];
M[MC,TZ[#2,#3,#4,#5,#6,#7] HMC[#1,ADD,#0]];

*MAKE AN SM CONSTANT WHICH IS NOT[SUM OF PARAMETERS]

M[NZ,ADD[NOT[#1],10000]];
M[NMC,TZ[#2,#3,#4,#5,#6,#7] HMC[#1,NZ,#0]];
```

```
%PUT A TRIPLE DISPATCH IN AN SM TABLE.  CALL IS SI[TABLE,DISP,D1,D2,D3];
WHERE TABLE AND DISP ARE INTEGERS.%

M[SI,(XSLC[E1[#3]E2[#4]E3[ADD[IP[#5],4000]]],SM[XSLC,ADD[#1,#2]]>];


%THE DISPATCH MEMORY SHOULD BE INITIALIZED AS FOLLOWS:

    FIRST DISPATCH IN D[31-43];
    SECOND DISPATCH IN D[15-27];
    THIRD DISPATCH IN D[1-13];
    MAIN-LOOP-BREAKOUT FLAG IN D[0].

THIS SHOULD BE DONE BY A MACRO CALL GIVEN IN THE PROGRAM LISTING
AFTER THE ROUTINES NAMED IN THE THREE DISPATCHES HAVE BEEN DEFINED,
SO THAT FORWARD REFERENCE FIXUPS WILL NOT BE REQUIRED FOR EACH OF
THE 512 PDP-10 OPCODES TIMES THREE.  TWO MACROS ARE PROVIDED FOR
THIS:  DI[OPCODE, 1ST, 2ND, 3RD] DOESN'T SET THE BREAKOUT FLAG;
DIS[OPCODE, 1ST, 2ND, 3RD] DOES.%

◆◆◆CHECK FOR INSTRUCTION MEMORY ADDRESSES IN EACH OF THESE ARGUMENTS??

M[DIS,(DLC[E1[#2] E2[#3] E3[#4]], DM[DLC,#1]>];
M[DI,(DLC[E1[#2] E2[#3] E3[ADD[IP[#4],4000]]], DM[DLC,#1]>];


◆SPECIAL STUFF FOR R MEMORY

M[RSRC,RA[#1]RB];
M[RSINK,RA[#1]RB←]; M[RB←]; M[RB←LU,LU]; M[RB←Q,(Q,AQ>];
M[R,#3#2#1R(RLC[#3#2#1R:E1[#1] E2[#2] E3[#3]],
    IFSE[#4,,,ER[#4#3#2#1R???]]>];   ◆CREATE LITERAL IN R
M[RV,RLC[#1:#2]];   ◆MAKE VARIABLE IN R
RM[RX,0]; RM[RAC,2];


◆SPECIAL STUFF FOR L MEMORY

M[LSRC,LA[#1]LB];
M[LSINK,LA[#1]LB←]; M[LB←]; M[LB←LU,LU]; M[LB←Q,(Q,AQ>];
M[L,#3#2#1L(LLC[#3#2#1L:E1[#1] E2[#2] E3[#3]],
    IFSE[#4,,,ER[#4#3#2#1L???]]>];   ◆CREATE LITERAL IN L
M[LV,LLC[#1: #2V]];
LM[LX,1]; LM[LINDX,0]; LM[LAC,3]; LM[LACS,2];
```

```
*F REGISTER PARAMETER DECLARATIONS

MP[OVF,0];        *OVERFLOW = (PC0#PC1) (HARDWARE DEPENDENT)
MP[PC0,1];        *ALU CARRY OUT OF BIT 0 (H.D.)
MP[PC1,2];        *ALU CARRY OUT OF BIT 1 (H.D.)
MP[FOVF,3];       *FLOATING OVERFLOW
MP[BYTEINT,4];    *INHIBIT BYTE POINTER INCREMENT ON NEXT ILDB OR IDPB
MP[UM,5];         *USER MODE (H.D.)
MP[PARCMODE,6];   *ENABLES USER MODE OPTIONS
MP[CFM,7];        *CALL-FROM-MONITOR.  DIRECTS FORCED REFERENCES TO USER
                  *ADDRESSES TO THE MONITOR (I.E., SUPPRESSES UMOVE, UXCT)

*10-12 RESERVED FOR MORE PROCESSOR FLAGS

MP[FUNF,13];      *FLOATING UNDERFLOW
MP[NODIV,14];     *NO DIVIDE
MP[PDOVF,15];     *PUSH-DOWN OVERFLOW
MP[XCT0,16];      *DIRECTS INDIRECT READS TO USER SPACE (H.D.)
MP[XCT1,17];      *READS AND RMW'S TO USER (H.D.)
MP[XCT2,20];      *BYTE POINTER INDIRECT READS TO USER (H.D.)
MP[XCT3,21];      *WRITES TO USER (H.D.)
SP[XCTN,XCT0,XCT1,XCT2,XCT3];
MP[INCOMPATIBLE,22];  *ENABLES PRIVILEGED OPTIONS
MP[PIACTIVE,23];   *P.I. SYSTEM TURNED ON
MP[PICYCLE,24];   *P.I. OR TRAP CYCLE IN PROGRESS
MP[MONHALT,25]; *MONITOR AFTER-LOADING TRAP (USED ONLY DURING MAP LOADING)
MP[THIRDPT,26]; *CHECKING THIRD P.T.E. DURING MAP LOADING
MP[MICRO,27];   *P.I. CHECKS SHOULD BE MADE

*30,31,32,34, AND 36 UNUSED.  MONALT AND THIRDPT MAY BE REUSED BY ROUTINES
*WHICH DON'T OVERLAP WITH MAP LOADING

MP[CUM,33];       *CURRENT USER MODE (H.D.)
MP[IENABLE,35]; *ENABLES MICROINTERRUPTS (H.D.)
MP[NOVA,37];      *NOVA HAS LEFT A MESSAGE (H.D.)
MP[K,40];         *(H.D.)
MP[J,41];         *(H.D.)
MP[H,42];         *(H.D.)
MP[G,43];         *(H.D.)


*PATCH MACRO FOR USE WITH MIDAS

M[E,(TARGET[ILC] IM[ILC,#1])];
```